C++学习笔记7

2022年3月19日 20:55

- 泛型算法
  - 概念
    - 大多数算法定义在头文件algorithm中，标准库还在头文件numeric中定义了一组数字的泛型算法
    - 一般情况下，这些算法不直接操作容器，而是遍历由两个迭代器指定的一个元素范围
  - 迭代器令算法不依赖于容器，但算法依赖于元素类型的操作
    - 泛型算法本身不会执行容器的操作，他们只会运行于迭代器之上，执行迭代器的操作。
- 泛型算法的分类
  - 只读算法
    - 一些算法只会读取其输入范围内的元素，而从不改变元素
      - find：找一个元素
      - count：计算元素的数目
      - accumulate：（在头文件numeric）：将范围内的元素求和
        - accumulate的第三个参数的类型决定了函数中使用那个加法运算符以及返回值的类型
    - 注意
      - 对于只读取而不改变元素的算法，通常最好使用cbegin（）和cend（）。但是如果你计划使用算法返回迭代器来改变元素的值，就需要使用begin（）和end（）作为参数
      -
  - 操作两个序列的算法
    - equal：用于确定两个序列是否保存相同的值
      - 此算法接受三个迭代器，前两个是范围，后一个是第二个序列的首元素
      - 注意
        - 只能比较两个相同的序列
        - 在比较两个char*（c风格字符串）时候会比较地址
  - 写容器元素的算法
    - fill：接受一个范围，将范围 内元素换成给定的元素
    - fill—n：
      - 接受一个范围，将范围内的元素换成给定的元素
      - 接受一个迭代器，两个数字，根据第一个数字将这个包括这个迭代器的数值换成后面那个迭代器的
    - back—inserter：插入迭代器（定义在iterator）
      - 插入迭代器是一种向容器中添加元素的迭代器

        - 与fill-n可以连用：传入一个插入迭代器插入一组相同的值
    - 注意：
        - 必须注意确保序列原大小至少不小于我们要求算法写入的元素数目
        - 向目的位置迭代器写入数据的算法假定目的位置足够大，能容纳要写入的元素
- 拷贝算法
    - 拷贝算法是另一个向目的位置迭代器指向的输出序列中的元素写入数据的算法
    - 此算法接受三个迭代器，前两个表示一个输入范围，第三个表示目的序列的起始位置
    - 此算法将输入范围内的元素拷贝到目的序列中。传递给copy的目的序列至少要包含与输入序列一样多的元素。这一点很重要
    - 多个算法都提供所谓的copy版本，
- 重排容器元素的算法
    - sort：调用其会重排容器中元素的顺序，使之有序（按字典顺序）
    - unique：调用其可以找出不重复的一部分
    - stable-sort：对于一个string，可以将其进行字典排序的同时将其按照长短排序
- 定制操作
    - 标准库还为这些算法定义了额外的版本，允许我们自己定义的操作来代替默认 的运算符
- 向算法传递函数
    - sort的第二个版本，重载函数
        - 谓词
            - 谓词是一个可以调用的表达式，其返回结果是一个能用作条件的值
            - 标准库算法所使用的谓词分为两类
                - 一元谓词：只接受一个参数的函数
                - 二元谓词：他们有两个参数的函数
            - 谓词很多都是bool类型，返回true或者false
    - find-if
        - find-if算法接受一对迭代器，表示一个范围，在第三个参数上是一个谓词，它返回一个使谓词返回非0值的元素，如果不存在这样的元素就返回尾迭代器
        -
    - lambda表达式
        - 形式
            - 一个lambda表达式表示一个可调用的代码单元。我们可以将其理解为一个未命名的内联函数。
            - 其中，capture list（捕获列表）是一个lambda所在函数中定义的局部变量的列表（通常为空）；return type、parameter list和function body与任何普通函数一样，分别表示返回类型、参数列表和函数体。但是，与普通函数不同，lambda必须使用尾置返回（参见6.3.3节，第206页）来指定返回类型。

$$[capture\ list](parameter\ \ list)\ \ ->\ \ return\ \ type\ \ \{\ \ function\ \ body\ \ \}$$

- 我们可以忽略参数列表和返回类型，但必须永远包含捕获列表和函数体
  auto f=【】{return 42};
- 在lambda中忽略括号和参数列表等价于指定一个空的参数列表
- 如果忽略返回类型，lambda会推断return语句的内容（如有的话），否则返回void
- 如果一个lambda体包含return之外的任何语句，则编译器假定此lambda返回void

□ 向lambda传递函数

- 使用【】捕获一些局部变量里面具体的数值用于表达

```
[sz](const string &a)
     { return a.size() >= sz; };
```

- 调用

```
// 获取一个迭代器，指向第一个满足 size()>= sz 的元素
auto wc = find_if(words.begin(), words.end(),
    [sz](const string &a)
        { return a.size() >= sz; });
```

- 传递的时候注意后续调用函数本身的性质，如果只能操作一个对象就只能传一个主要的，如果能操作两个就能传两个
- 在调用lambda的时候，记得加括号

```
auto y=[sz]()mutable->bool{auto yy=sz;while(sz>0)sz--;if(yy>0)return true;else return false;};
cout<<y();
```

□ lambda的捕获和返回

- 当定义一个lambda时候，编译器生成一个与lambda对应的新的类类型
- 可以这样理解：当向一个函数传递lambda时，同时定义了一个新的类型和该类型的一个对象，类似的，当使用auto定义了一个用lambda初始化的变量时，定义了一个从lambda生成的类型和对象
- lambda的数据成员也在lambda对象创建时初始化
- 捕获方式
  - ◇ 值捕获
    - ▸ 类似于参数传递，采用值捕获的前提使变量可以拷贝
    - ▸ 与参数不同，被捕获的变量的值是在lambda创建时拷贝，而不是调用时拷贝
    - ▸ 由于被捕获的变量的值实在lambda创建时拷贝，因此随后对其修改不会影响到lambda内对应的值
  - ◇ 引用捕获
    - ▸ 对于一些不能拷贝的对象：ostream我们可以使用引用捕获
    - ▸ 当以引用捕获一个变量的时候，我们必须保证在lambda执行的时候变量是存在的
  - ◇ 隐式捕获
    - ▸ 为了指示编译器推断捕获列表，应在捕获列表中写一个&或=

- &：表示告诉编译器采用捕获引用方式
- =：表示采用值捕获方式
- 可以混合使用值捕获和隐式捕获的方式（【&，=】）

◇ 操作

表 10.1：lambda 捕获列表

[]	空捕获列表。lambda 不能使用所在函数中的变量。一个 lambda 只有捕获变量后才能使用它们
[*names*]	*names* 是一个逗号分隔的名字列表，这些名字都是 lambda 所在函数的局部变量。默认情况下，捕获列表中的变量都被拷贝。名字前如果使用了&，则采用引用捕获方式
[&]	隐式捕获列表，采用引用捕获方式。lambda 体中所使用的来自所在函数的实体都采用引用方式使用
[=]	隐式捕获列表，采用值捕获方式。lambda 体将拷贝所使用的来自所在函数的实体的值
[&, *identifier_list*]	*identifier_list* 是一个逗号分隔的列表，包含 0 个或多个来自所在函数的变量。这些变量采用值捕获方式，而任何隐式捕获的变量都采用引用方式捕获。*identifier_list* 中的名字前面不能使用&
[=, *identifier_list*]	*identifier_list* 中的变量都采用引用方式捕获，而任何隐式捕获的变量都采用值方式捕获。*identifier_list* 中的名字不能包括 this，且这些名字之前必须使用&

◆ 建议

◇ 尽量保持lambda的变量捕获简单化

▸ 如果 我们捕获一个指针或迭代器，或采用引用捕获的方式，就必须确保在lambda执行的时，绑定到迭代器，指针或引用的对象依然存在且要保证对象由预期的值

▸ 一般来说，我们应该尽量减少捕获的数据量，来避免潜在的捕获导致的问题，而且，如果可能的话，应该避免指针或者引用

□ 可变lambda

◆ 默认情况下，对于一个值被拷贝的变量，lambda不会改变其值

◆ 如果我们希望改变一个被捕获变量的值，就必须在参数列表首加上关键词mutable

◇ 可变lambda能省去参数列表

◇ 值捕获

```
void fcn3()
{
    size_t v1 = 42; // 局部变量
    // f 可以改变它所捕获的变量的值
    auto f = [v1] () mutable { return ++v1; };
    v1 = 0;
    auto j = f(); // j 为 43
}
```

◇ 引用捕获

```
void fcn4()
{
    size_t v1 = 42; // 局部变量
    // v1 是一个非 const 变量的引用
    // 可以通过 f2 中的引用来改变它
    auto f2 = [&v1] { return ++v1; };
    v1 = 0;
    auto j = f2(); // j 为 1
}
```

- 指定lambda的返回类型
  - 如果一个lambda体包含return之外的任何语句，则编译器假定此lambda返回void
  - 所以如果括号里执行多条语句且需要返回，必须指定返回类型

```
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) -> int
          { if (i < 0) return -i; else return i; });
```

- 标准库的bind函数
  - 它定义在functional中，可以将bind函数看作一个通用的函数适配器，它接受一个可调用对象，生成一个新的可调用对象来适应原对象的参数列表
  - 我们利用占位符表示需要调用的对象，多余的位置表示顺便要带进去的元素

```
// check6 是一个可调用对象，接受一个 string 类型的参数
// 并用此 string 和值 6 来调用 check_size
auto check6 = bind(check_size, _1, 6);

string s = "hello";
bool b1 = check6(s); // check6(s)会调用 check_size(s, 6)
```

    - 在调用bind时候也是一样的，因为它返回一个整体的函数，所以调用的时候要注意

```
int x=count_if(a.begin(),a.end(),bind(isShorter,_1,6));
```

  - 对于_n，我们要使用名字placeholders（using namespace placeholders）
  - bind的参数（值引用参数绑定）
    - 假定f是一个可以调用的对象，他可以对被调用的参数重新安排顺序

```
f(a, b, Y, c, X)
```

      例如对g（X，Y）的调用会变为
  - bind的参数（引用参数绑定）
    - 我们需要使用一个ref函数，ref返回一个对象，包含给定的引用，此对象是可以拷贝的，cref函数返回一个const引用的类，与bind一样，函数ref和cref都定义在functional中

        - 建议
            - 对于那种只在一两个地方使用的简单操作，lambda表达式是最有用的，如果我们需要在很多地方使用相同的操作，通常应该定义一个函数，而不是多次编写相同的lambda表达式。
            - 如果一个操作需要很多语句才能完成，通常使用函数更好
            - 如果lambda的捕获列表为空，通常可以使用函数来代替他
- 迭代器与泛型算法（再探迭代器）
- 头文件iterator中定义了额外几种迭代器
    - 插入迭代器：这些迭代器被绑定到一个容器上，可用来向容器中插入元素
        - back—inserter：创建一个使用push-back的迭代器
        - front—inserter：创建一个使用push-front的迭代器
        - inserter：创建一个使用insert的迭代器，此函数接受第二个参数，这个参数必须是一个指向给定容器的迭代器，元素将被插入到给定迭代器所表示的元素之前
        - 注意
            - 只有在支持push-front，push-back情况下才能使用
    - 流迭代器：这些迭代器被绑定到输入或者输出的流上，可用来遍历所关联的IO流
    - 反向迭代器：这些迭代器向后而不是向前移动。除了forward—list之外的标准库容器都有反向迭代器
    - 移动迭代器：这些专用的迭代器不是拷贝它们而是移动他们
    - iostream迭代器：标准库定义了可以用于这些IO类型对象的迭代器
        - 这些迭代器将他们对应的流当作一个特定类型的元素序列来处理，通过使用流迭代器，我们可以用泛型算法从流对象读取数据以及向其写入数据
        - istream-iterator：读取输入流
            - 操作：一个istream-iterator使用》来读取流。因此迭代器要读取的类型必须定义了输入运算符，只要有输入运算符就可以使用istream-iterator

```
istream_iterator<int> int_it(cin);    // 从 cin 读取 int
istream_iterator<int> int_eof;        // 尾后迭代器
ifstream in("afile");
istream_iterator<string> str_it(in); // 从"afile"读取字符串
```

表 10.3：istream_iterator 操作

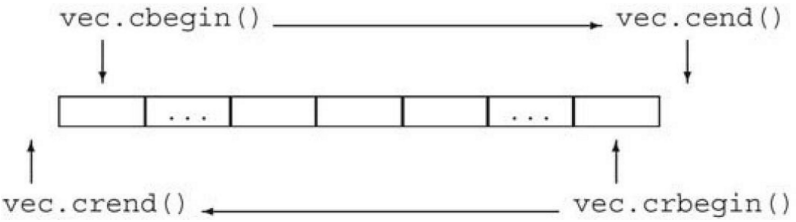
istream_iterator<T> in(is);	in 从输入流 is 读取类型为 T 的值
istream_iterator<T> end;	读取类型为 T 的值的 istream_iterator 迭代器，表示尾后位置
in1 == in2 in1 != in2	in1 和 in2 必须读取相同类型。如果它们都是尾后迭代器，或绑定到相同的输入，则两者相等
*in	返回从流中读取的值
in->mem	与 (*in).mem 的含义相同
++in, in++	使用元素类型所定义的>>运算符从输入流中读取下一个值。与以往一样，前置版本返回一个指向递增后迭代器的引用，后置版本返回旧值

            - 空的istream-iterator可以当作尾后迭代器使用
        - ostream-iterator：向一个输出流写数据
            - 操作
                - 一个0stream-iterator使用《来输出流。因此迭代器要读取的类型必须定义了输出运算符，只要有输出运算符就可以使用ostream-iterator

表 10.4：ostream_iterator 操作

`ostream_iterator<T> out(os);`	`out` 将类型为 `T` 的值写到输出流 `os` 中
`ostream_iterator<T> out(os,d);`	`out` 将类型为 `T` 的值写到输出流 `os` 中，每个值后面都输出一个 `d`。`d` 指向一个空字符结尾的字符数组
`out = val`	用<<运算符将 `val` 写入到 `out` 所绑定的 `ostream` 中。`val` 的类型必须与 `out` 可写的类型兼容
`*out, ++out, out++`	这些运算符是存在的，但不对 `out` 做任何事情。每个运算符都返回 `out`

        - 注意
            - iter进行读取输出操作的时候，与io一样，会有相类似的缓冲和跳过的机制
    - 反向迭代器：从尾元素向首元素反向移动的迭代器
        - 反向迭代器也提供反向的c类型操作符使用
        - 对于反向迭代器，递增和递减操作和原来的交换，打印迭代器中的内容的时候一样会逆序



        - 如果要将反向迭代器变回正序，需要使用base函数

```
//正确：得到一个正向迭代器，从逗号开始读取字符直到 line 末尾
cout << string(rcomma.base(), line.cend()) << endl;
```

- 泛型算法的结构
    - 输入迭代器：只读不写，单遍扫描，只能递增
        - 支持运算符
            - 相等或不相等运算符==/！=
            - 递增运算符++
            - 读取元素，解引用运算符*
            - 箭头运算符-》/（*xx）。xxx
        - 输入迭代器只用于顺序访问，对于一个输入迭代器，*it++保证是有效的，但递增他可能导致其他所有指向流的迭代器失效
        - 常见的输入迭代器：istream-iterator
    - 输出迭代器：只写不读，单边扫描，只能递增
        - 支持运算符
            - 递增运算符++
            - 解引用运算符（*）
        - 我们只能向一个输出迭代器赋值一次，类似输入迭代器，输出迭代器只能用于单遍扫描算法
        - 常见：ostream-iterator
    - 前向迭代器：可读写，多遍扫描，只能递增
        - 支持
            - 所有输入输出迭代器的操作
            - 多次读写一个元素

        - 只能沿着一个方向移动
    - 常见：replace，forward-list
  - 双向迭代器：可读写，多遍扫描，可递增递减
    - 支持
      - 可以正向/反向读写序列中的元素
      - 支持所有前向迭代器的操作
      - 支持递减运算符
    - 常见：除了forward-list其他的都支持
  - 随机访问迭代器：可读写，多遍扫描，支持全部迭代器运算：全能
    - 常见：vector，array，deque，string，访问内置数组的指针
- 算法的形参模式
  - 大多数算法具有如下的几种形式之一
    - 操作

```
alg(beg, end, other args);
alg(beg, end, dest, other args);
alg(beg, end, beg2, other args);
alg(beg, end, beg2, end2, other args);
```

      - dest
        - dest参数是一个可以表示算法可以写入的目的位置的迭代器，算法假定按需写入数据，不管写入多少个元素都是安全的
          - 向输出迭代器写入数据的算法都假定目标空间足够容纳写入的数据
        - dest可能会被绑定一个插入迭代器或是一个ostream-iterator
      - 接受第二个输入序列的算法
        - 接受beg2，end2的算法表示第二个输入范围，这些算法通常使用第二个范围中的元素与第一个输入范围结合来进行一些运算
        - 如果同时接受beg2，end2，表示2个范围：【beg，end），【beg2，end2）
        - 接受单独beg2的算法假定从beg2开始的序列与beg和end所表示的范围至少一样大
- 算法的命名规范
  - 算法还遵循一套命名规范和重载规范。
  - 一些算法使用重载形式传递要给谓词
    - 接受谓词参数来代替<或==的算法，以及哪些不接收额外参数的算法，通常都是重载的函数

```
unique(beg, end);         // 使用 == 运算符比较元素
unique(beg, end, comp);   // 使用 comp 比较元素
```

  - -if版本的算法
    - 接受一个元素值的算法通常由另一个不同名（不是重载的）版本，该版本接受一个谓词代替元素值

        - 接受谓词参数的算法都有附加的-if前缀

```
find(beg, end, val);          // 查找输入范围中 val 第一次出现的位置
find_if(beg, end, pred);      // 查找第一个令 pred 为真的元素
```

    - 区分拷贝元素的版本和不拷贝的版本
        - 默认情况下，重排元素的算法将重拍后的元素写回给定的输出序列中，这些算法还提供另外一个版本，将元素写道一个指定的输出目的位置
        - 一些算法会同时提供-if和-copy版本。这些版本接受一个目的位置迭代器和一个谓词

```
reverse(beg, end);                // 反转输入范围中元素的顺序
reverse_copy(beg, end, dest);     // 将元素按逆序拷贝到 dest
```

- 特定容器算法（list，forward-list）
    - 与其他成员不同，list和forward-list定义了几个成员函数算法
    - 他们有独有 的sort，merge，remove，reverse，unique；
    - 有些算法的通用版本可以用于链表，但是代价高且不快
    - 对于list和forward-list，应该优先使用成员函数版本的算法而不是通用算法
    - list特殊版本算法
        - 表 10.6：list 和 forward_list 成员函数版本的算法

这些操作都返回 **void**	
lst.merge(lst2)	将来自 lst2 的元素合并入 lst。lst 和 lst2 都必须是有序的。元素将从 lst2 中删除。在合并之后，lst2 变为空。第一个版本使用<运算符；第二个版本使用给定的比较操作
lst.merge(lst2, comp)	
lst.remove(val)	调用 erase 删除掉与给定值相等（==）或令一元谓词为真的每个元素
lst.remove_if(pred)	
lst.reverse()	反转 lst 中元素的顺序

lst.sort()	使用<或给定比较操作排序元素
lst.sort(comp)	
lst.unique()	调用 erase 删除同一个值的连续拷贝。第一个版本使用==；第二个版本使用给定的二元谓词
lst.unique(pred)	

    - splice算法（链表特有）
        - 操作
            - 与merge不同，splice会直接销毁给定的链表，但是merge之后两个链表还是存在的
    - 注意
        - 对于list，forward-list，调用算法会真正的改变容器的大小
-
    - 注意有些算法是需要提供随机访问迭代器才可以用的，如list中的sort